Recall Minimum Spanning Tree Problem.

Last day:
Kruskal's Algorithm & implementation \(O(m \log n)\)

**Prim's Algorithm.**
Grow one connected component in a greedy fashion (i.e. by adding min. weight edge leaving the component)

\[ C \subseteq V \]

Choose min. weighted edge leaving \( C \)

\( C = \) set of vertices reached by \( T \) so far

Initialize \( C \leftarrow \emptyset \), \( T \leftarrow \emptyset \)

While \( C \neq V \)

Find min. weight edge \( e = (u, v) \) from

\( u \in C \) to \( v \in V \setminus C \)

\( T \leftarrow T \cup \{ e \} \)

\( C \leftarrow C \cup \{ v \} \)

End

Correctness: The exact same exchange argument works. And in fact, we could prove one lemma that gives correctness of both alg.s. (See text.)
Prim – implementation.

In general, we need to find min weight edge leaving \( C \), the connected component of \( T \).

Priority Queue data structure

Maintain set of weighted elements (in our case) edges leaving \( C \)

Operations

- Find and delete min weight element
- Insert
- Delete

Can be implemented as a heap (see CS 240 or text) at \( O(\log k) \) time per operation, \( k = \# \) elements

In our case

\[ \delta(C) = \text{edges leaving } C \]

Changes to \( \delta(C) \) when \( v \) is added to \( C \):

- Edges from \( C \) to \( v \) leave \( \delta(C) \)
- Other edges adjacent to \( v \) enter \( \delta(C) \)

We can find these edges by going through \( v \)’s adjacency list.

Each edge enters \( \delta(C) \) once and leaves it once.
Priority Queue operations

- n-1 find min
- m insert
- m delete

Total cost:

\[ O(n \log m + m \log m) = O(m \log n) \]

\[ \uparrow \uparrow \]

Find, Insert, Delete

It is slightly more efficient to keep a priority queue of vertices \( V \sim C \) with weight \( \text{weight}(v) = \text{min weight edge from } C \text{ to } v \)

- size of PQ = \( n \)
- update is key-change \( O(\log n) \)
- still gives \( O(m \log n) \) total.

Additional improvement

- use Fibonacci heap to implement PQ
- then decrease key is \( O(1) \)

so we get

\[ O(n \log n + m) = O(m + n \log n) \]

\[ \text{find} \quad \text{decrease key} \quad (\text{key change}) \]
Shortest Paths in Edge Weighted Graphs

Recall that BFS from \( v \) finds shortest paths from \( v \) in unweighted undirected graphs.

General input: directed graph with weights on edges.

Note: An represent undirected edge as 2 directed edges

\[
\begin{align*}
  \text{Shortest path } A - D \text{ is } ABD, \text{ weight 5.} \\
  A - E \quad ABE, \text{ weight 4.}
\end{align*}
\]

We will sometimes allow negative weights but we'll assume no negative weight cycle (otherwise go around it \( \infty \) to get \( -\infty \) length.)

[Note: we might still want a shortest path that is simple (doesn't repeat vertices) but that's NP-complete)

Versions of the problem:

1. Given \( u, v \), find shortest \( u \rightarrow v \) path
2. Given \( u \), find shortest \( u \rightarrow v \) path \( \forall v \)
   "single source shortest path problem"
3. Find shortest \( u \rightarrow v \) path \( \forall u, v \)
   "all pairs shortest path problem"

Solving 1 seems to involve solving 2.
But we can solve 2 faster than 3.
Start with 2. Do 3 later (dynamic programming)
Single Source Shortest Paths in Directed Graphs
- general weights (but no neg. cycle) \( O(mn) \)
  - Bellman Ford
- no cycles \( O(n+m) \)
- no negative weights \( O(m \log n) \)
  - Dijkstra's algorithm

**Dijkstra's Algorithm** 1959

**Input**: digraph \( G = (V,E) \), \( w : E \to \mathbb{R}^\geq \), \( s \in V \)
  - non-neg edge weights
  - source

**Output**: shortest path from \( s \) to every other vertex \( v \).

**Idea**: Grow tree of shortest paths starting from \( s \)

**General step**: have tree of shortest paths to all vertices in set \( B \)

Initially \( B = \{s\} \)

Choose edge \( (x, y) \in B \), \( y \notin B \)

to minimize \( d(s, x) + w(x, y) \)

\( d(s, y) \leftarrow d \)

add \( (x, y) \) to tree (Parent(y) \( \leftarrow x \) )

Note similarity & differences to Prim's MST alg.
This is greedy in the sense that we always add the vertex with next min distance from s.

Claim: \( d \) is the min. distance from \( s \) to \( y \).

[This justifies the output being a tree]

Proof: Any path \( \Pi \) from \( s \) to \( y \) consists of
- \( \Pi_1 \): initial part of path in \( B \)
- \( (u,v) \): first edge leaving \( B \)
- \( \Pi_2 \): rest of path.

\[
\begin{align*}
    w(\Pi) &\geq w(\Pi_1) + w((u,v)) \\
    &\geq d(s,u) + w(u,v) \\
    &\geq d
\end{align*}
\]

using that \( w(\Pi_2) \geq 0 \)

- the proof breaks down for neg. weight cycles

Therefore, by induction on \( |B| \), the alg. correctly finds \( d(s,v) \) for all \( v \).

Implementations

- want to choose edge leaving \( B \) to minimize some value
- could make a heap of edges \( (x,y) \in E \), \( x \in B \), \( y \notin B \)
  where \( value(x,y) = d(s,x) + w(x,y) \)
  This heap has size \( O(n) \)
- more efficient: a heap of vertices
Keep "tentative distance" \( d(v) \) \( \forall v \neq s \)
\( d(s) = 0 \)
\( B = \emptyset \)

While \( |B| < n \)

1. \( y \leftarrow \text{vertex of } V \setminus B \text{ with min. } d \text{ value - from heap} \)
2. \( B \leftarrow B \cup \{y\} \) -- note that \( d(y) \) is true distance

For each edge \((y,z)\) do

1. if \( d(y) + w(y,z) < d(z) \) then
2. \( d(z) \leftarrow d(y) + w(y,z) \) -- and update heap
3. \( \text{Parent}(z) \leftarrow y \)

End

End

Store \( d \) values in a heap.  Size is \( \leq n \)

Modifying a \( d \) value takes \( O(\log n) \) to adjust heap.

Total time \( O(n \log n + m \log n) = O(m \log n) \)

Actually, there is a fancier "Fibonacci heap" that gives \( O(n \log n + m) \)  see CLRS